From Event-B Models to Dafny Code Contracts

Mohammadsadegh Dalvandi, Michael Butler, Abdolbaghi Rezazadeh

Electronic and Computer Science School, University of Southampton
Southampton, United Kingdom
{md5g11,mjb,ra3}@ecs.soton.ac.uk

Abstract. The constructive approach to software correctness aims at formal modelling and verification of the structure and behaviour of a system in different levels of abstraction. In contrast, the analytical approach to software verification focuses on code level correctness and its verification. Therefore it would seem that the constructive and analytical approaches should complement each other well. To demonstrate this idea we present a case for linking two existing verification methods, Event-B (constructive) and Dafny (analytical). This approach combines the power of Event-B abstraction and its stepwise refinement with the verification capabilities of Dafny. We presented a small case study to demonstrate this approach and outline of the rules for transforming Event-B events to Dafny contracts. Finally, a tool for automatic generation of Dafny contracts from Event-B formal models is presented.

Keywords: Event-B, Dafny, Formal Methods, Program Verification, Methodologies

1 Introduction

The constructive approach to software correctness focuses on early stages of the development and aims at formal modelling of the intended behaviour and structure of a system in different levels of abstraction and verifying the formal specification of it. In contrast, the analytical approach focuses on code level and its target is to verify the properties of the code level. In other words, the constructive approach is concerned with the derivation of an algorithm from the specifications of the desired dynamic behaviour of that, in a way that the algorithm satisfies its specification [5] while the analytical approach is concerned with verifying that a given algorithm satisfies its given specifications. Both approaches are supported through a range of verification tools from groups worldwide. At a high level it would seem that the constructive and analytical approaches should complement each other well. However there is little understanding or experience of how these approaches can be combined at a large scale and very little tool support for transitioning from constructive formal models to annotated code that is amenable to analytical verification. This represents a wasted opportunity, as deployments of the approaches are not benefiting from each other effectively.

This paper presents work in progress on a tool-supported development approach by linking two existing verification tools, Rodin [2] and Dafny [4]. The

Rodin platform supports the creation and verification of Event-B formal models. The Dafny tool is an extension to Microsoft Visual Studio for writing and verifying programs written in the Dafny programming language. Event-B in its original form does not have any support for the final phase of the development(implementation phase). On the other hand, Dafny has a very little support for abstraction and refinement. Our combined methodology is beneficial for both Event-B and Dafny users. It makes the abstraction and refinement of Event-B available for generating Dafny specifications which are correct with regards to a higher level of abstract specification in Event-B and allows Event-B models to be implemented and verified in a programming language. We discuss our approach for transforming Event-B formal models to annotated Dafny method declarations. Our focus here is only on generating code contracts (pre- and post-conditions) from Event-B models rather than implementations. Generated method contracts with this approach can be seen as an interface that can be implemented and verified later against the high level abstract specification. We also present a tool for automatic generation of Dafny annotations from Event-B models. We have validated our transformation rules by applying our tool to an Event-B model of a map abstract datatype which is presented in this paper.

The organisation of the rest of the paper is as follows: in section 2, background information on Event-B and Dafny is given. Section 3 contains an example of transformation of an Event-B model of a map abstract datatype to Dafny contracts. Transformation rules for transforming an Event-B machine to an annotated Dafny class are described in section 4. In section 5 related and future work are presented and finally section 6 contains conclusions.

2 Background

2.1 Event-B

Event-B is a formal modelling language for system level modelling based on set theory and predicate logic for specifying, modelling and reasoning about systems, introduced by Abrial [1]. Modelling in Event-B is facilitated by a platform called Rodin [2]. Rodin is an extensible open source software which is built on top of the Eclipse IDE. A model in Event-B consists of two main parts: *contexts* and *machines*. The static part (types and constants) of a model is specified in a context and the dynamic part (variables and events) is specified in a machine. To describe the static part of a model there are four elements in the structure of a context: *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are represented by their name and they are distinct from each other. Constants are defined using axioms. Axioms are predicates that express properties of sets and constants. Theorems in contexts can be proved from axioms. A machine in Event-B consists of three main elements: (1) a set of *variables*, which defines the states of a model (2) a set of *invariants*, which is a set of conditions on state variables that must hold permanently by all events and (3) a number of *events* which model the state change in the system. Each event may have a number of assignments called actions and also may have a number of guards. Guards are

predicates that describe the necessary conditions which should be true before an event can occur. An event may have a number of parameters. Event parameters are considered to be local to the event. Figure 1 illustrates machine *m0* with two events *Add* and *Remove*.

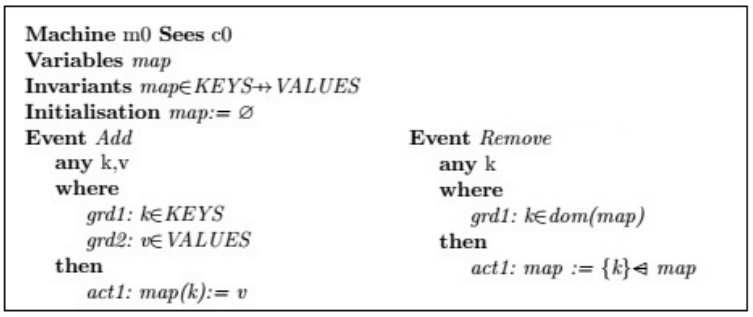

Fig. 1. Machine *m0*: the Most Abstract Level of Map ADT Model

Modelling a complex system in Event-B can benefit from refinement. Refinement is a stepwise process of building a large system starting from an abstract level towards a concrete level [1]. This is done by a series of successive steps in which, new details of functionality are added to the model in each step. The abstract level represents key features and the main purpose of the system. Refining an Event-B machine may involve adding new events or new variables(concrete variables). Concrete variables are connected to abstract variables through *gluing invariants*. A gluing invariant associates the state of the concrete machine with that of its abstraction. All invariants of a concrete model including gluing invariants should preserve by all events. The built-in mathematical language of the Rodin platform is limited to basic types and constructs like integers, boolean, relations and so on. The *Theory Plug-in* [3] has been developed to make the core language extension possible. A theory, which is a new kind of Event-B component, can be defined independently from a particular model and it is the mean by which the mathematical language and mechanical provers may be extended.

2.2 Dafny

Dafny [4] is an imperative sequential programming language which supports generic classes, dynamic allocation and inductive datatypes and has its own specification constructs. A Dafny program may contain both specification and implementation details. Specifications are omitted by the compiler and are used just during the verification process. Programs written and specified in Dafny can be verified using the Dafny verifier which is based on an SMT-solver. Standard pre- and post-conditions, framing construct and termination metrics are included in the specifications. The language offers updatable ghost variables,

recursive functions, sets, sequences and some other features to support specification. The verification power of Dafny originates from its annotations (contracts). A program behaviour can be annotated in Dafny using specification constructs such as methods' pre- and post-conditions. The verifier then tries to prove that the code behaviour satisfies its annotations. This approach leads to producing correct programs not only in terms of syntax but also in terms of behaviour. A basic program in Dafny consists of a number of methods. A method in Dafny is a piece of imperative, executable code. Dafny also supports *functions*. A function in Dafny is different from a method and has very similar concept to mathematical functions. A Dafny function cannot write to memory and consists of just one expression. A special form of functions which returns a boolean value is called *predicate*. Dafny uses the `ensures` keyword for post-condition declaration. A post-condition is always a boolean expression. Each method can have more than one post-condition which can either be joined with boolean *and* (&&) operator or be defined separately using the `ensures` keyword. To declare a pre-condition the `requires` keyword is used. Like post-conditions, adding multiple pre-conditions is allowed in the same style. Pre- and post-conditions are placed after method declarations and before method bodies. Dafny does not have any specific construct for specifying class invariants. Class invariants are specified in a predicate named *Valid()* and this predicate is incorporated in all methods pre- and post-conditions so the verifier checks if each method preserve all invariants or not.

3 Case Study: A Map Abstract Data Type

In this section we present a map abstract datatype as a case study and show the Event-B formal model and its transformation to Dafny contracts that is performed by our tool. A map (also called associated array) is an abstract data type which associates a collection of unique keys to a collection of values. This case study is originally taken from [6] where the map ADT is specified, implemented and verified in Dafny. The most abstract model of the map in Event-B (machine *m0*) is illustrated in Figure 1. The map is simply modelled using a partial function from *KEYS* to *VALUES*. *KEYS* and *VALUES* are generic types which are defined in a context (not shown here) as carrier sets. There is only one invariant in this model which says that the variable *map* is a partial function. The model contains two events for add and removing keys and values to the map. By proving that these events preserve the invariant of the model, the uniqueness of the map's keys is verified.

Dafny does not support relations (and functions) as data structures so we cannot directly transform machine *m0* to Dafny annotations. Machine *m0* should be refined in order to reduce the abstraction and syntax gap between the Event-B model and Dafny specification. In the refined machine two new variables *keys* and *values* are introduced to model. Variable *keys* is a sequence of type *KEYS* and variable *values* is a sequence of type *VALUES*. Sequences are built-in data structures in Dafny but they are not part of the built-in mathematical language

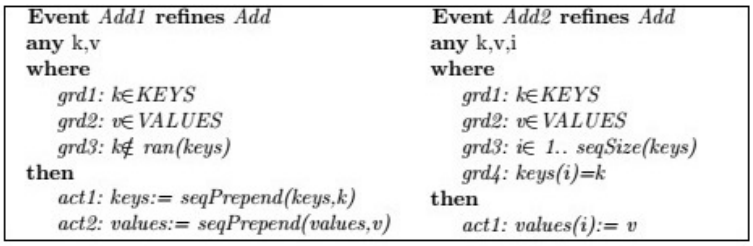

Fig. 2. Event *Add* is refined to two events *Add1* and *Add2*

of Rodin. However sequences are available through the standard library of the Rodin theory plug-in. As the name suggests sequence *keys* stores keys and the other sequence stores values where a value in position i of sequence *values* is associated with the key that is stored in position i of sequence *keys*. An invariant is needed in this refinement to state that both sequences have the same size. A gluing invariants is also needed to prove the consistency between refinements. Figure 2 shows that the event *Add* from machine *m0* is refined to two events *Add1*(for adding new keys to the map) and *Add2*(for updating an associated value to an existing key). Refinement of event *Remove* and other elements of the refined machine are omitted here because of the space limitation. Listing 1.1 shows the transformation of events of the refined machine to an annotated Dafny method called *Add*.

```
method Add(k:KEYS, v:VALUES)
requires Valid();
ensures (k !in old(keys) && keys==[k] + old(keys) &&
values==[v] + old(values))
||
(exists i :: i in (set k0| 0<=k0 && k0<|old(keys)|) &&
old(keys)[i]==k && values==old(values)[i:= v] &&
keys == old(keys));
```

Listing 1.1. Transformation of Machine *m1* to a Dafny Contract

Post-conditions of the *Add* method are directly derived from those events that form the method and they specify the behaviour of the method. Method *Add* is specified by two events in Event-B therefore two `ensures` clauses are generated (beside `ensures` *Valid();*). The reason for specifying a method with two Event-B events is that each event represents a separate case of the method and each case in a Dafny method is represented with a separate post-condition in the method contract. The keyword `old` which is used in the post-conditions of methods represents the value of the variable on entry to the method. Internal variables of each event are defined using existential quantifier with regards to the event's guards. The class declaration, predicate *Valid()* and other details of the generated class are not shown here. The transformation of Event-B events to annotated Dafny methods is discussed in the next section.

4 Transforming Event-B Models to Dafny Contracts

In this section we describe how we generate Dafny contracts from Event-B events. In order to be able to merge different Event-B events together to form a single method from them in Dafny, we have introduced a new element to Event-B machines called *constructor statement*. A constructor statement has the following form:

`method` *mtd_name*(*pi_1, pi_2,...*) `returns`(*po_1, po_2,...*) {*evt_1, evt_2,...*}

In the above statement, *mtd_name* is the name of the target method in Dafny, *(pi_1, pi_2,...)* represents the list of input parameters, *(po_1, po_2,...)* represents the list of output parameters, and *evt_1, evt_2,...* represents the list of Event-B events that must be merged together to form the target method.

A method may or may not have input/output parameters. Input parameters which are stated in the constructor statement must exist in all events which are listed in the the statement and also the type of the parameters must be explicitly declared in Event-B events as guards of the event. If a method in a constructor statement has a parameter which is not listed as a method's input/output parameter, it should be treated as an *internal parameter*. An internal parameter is a local variable to the method and will be specified using an existential quantifier. A number of post-conditions can be generated from *before-after* predicates of the actions of the events together with their guards. A before-after predicate denotes the relation that exists between the value of a variable just before and just after the execution of an action. In the example shown in the previous section, the method *Add* was generated as a result of the following constructor statement:

`method` *Add*(k,v) `returns`() {*Add1*, *Add2*}

Consider *act1* of event *Add1* from Figure 2. The before-after predicate associated with this action is $keys' = seqPrepend(keys, k)$ where the primed variable denotes the value of the variable just after the execution and the unprimed variables denote the value of the variables before the execution. The following expression can be derived from event *Add1* by conjunction of all non-typing guards of the event before-after predicates of all actions of the event. The result would be for event *Add1*:

$$k \notin ran(keys) \wedge keys' = seqPrepend(keys, k) \wedge values' = seqPrepend(values, v) \quad (1)$$

The same should be done for event *Add2*. As it is obvious from the action of event *Add2*, variable *keys* is not changed by this event therefore the value of this variable after the execution of the event is equal to its value before the execution. The following expression is derived from this event:

$$i \in 1..seqSize(keys) \wedge keys(i) = k \wedge values'(i) = v \wedge keys' == keys \quad (2)$$

Note that event *Add2* has a third parameter i which is not listed as *Add* method parameter in constructor statement so it is an internal parameter and should be specified using an existential quantifier:

$$\exists i \cdot i \in 1..seqSize(keys) \wedge keys(i) = k \wedge values\prime(i) = v \wedge keys\prime == keys \quad (3)$$

The disjunction of (1) and (3) becomes the post-condition for method *Add* and specifies the desirable behaviour of the method. In addition to the generated contracts from events of the Event-B model, predicate *Valid()*(which contains the conjunction of machines invariants) must be a pre-condition for all method declarations. This is necessary as the verifier needs this information to be able to verify the post-conditions.

4.1 Tool Support for Automatic Transformation

We have developed a Rodin plug-in for automatic transformation of Event-B machines to annotated Dafny classes. The plug-in builds an abstract syntax tree (AST) with regards to the Event-B machine and contexts that it sees and constructor statements that are provided by the user. The AST then is translated to Dafny code by a number of translation rules that are encoded in the plug-in source code. The tool only supports the translation of those Event-B mathematical constructs that have a counterpart in Dafny and ignores the rest. So it is important that the model should be refined to a level that only has those constructs that have a Dafny counterpart.

5 Related and Future Work

To the best of our knowledge, no research has been carried out in order to generate annotated Dafny programs from Event-B models and there is very little research on generating verifiable code from Event-B models. EventB2Dafny [7] is a Rodin plug-in for translating Event-B proof obligations to Dafny code to use Dafny verifier as an external theorem prover for proving Event-B proof obligations. Another research has been carried out in order to translate Even-B models to JML-specified JAVA code. A Rodin plug-in called EventB2JML [8] has been developed to automate the translation from Event-B models to Java specified code. Tasking Event-B [9] is a code generator that generates code from Event-B models to a target language but it does not support verification of the generated code.

Our current transformation rules allow us to generate Dafny contracts for abstract data types. We plan to extend our rules and the tool to be able to generate code contracts from Event-B model of complex algorithms. We have already done another case study for transforming an Event-B model of a model checking algorithm to Dafny contracts.

6 Conclusion

We have presented an approach for generating Dafny code contracts from Event-B models. This approach allows us to start the development with a very high level specification of the program in Event-B and use the Rodin platform facilities to prove the correctness and consistency of specification and refine the specification to a level that is suitable for transformation to Dafny. The implementation can be done later manually and verified against the abstract specification. The abstraction level that can be achieved in a modelling language like Event-B is not achievable at Dafny level therefore using the stepwise manner of Event-B for building specification will help to tackle the complexity that is associated with this task.

Acknowledgments. This work was funded in part by a Microsoft Research 2014 Software Engineering Innovation Foundation Award.

References

1. Abrial, J. R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010)
2. Abrial, J. R., Butler, M., Hallerstede, S., Hoang, T. S., Mehta, F., & Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International journal on software tools for technology transfer, 12(6), 447-466. (2010)
3. Butler, M., & Maamria, I.: Practical theory extension in Event-B. In Theories of Programming and Formal Methods (pp. 67-81). Springer Berlin Heidelberg. (2013)
4. Leino, K. R. M.: Dafny: An automatic program verifier for functional correctness. In Logic for Programming, Artificial Intelligence, and Reasoning (pp. 348-370). Springer Berlin Heidelberg. (2010)
5. Dijkstra, E. W.: A constructive approach to the problem of program correctness. BIT Numerical Mathematics, 8(3), 174-186. (1968)
6. Leino, K. R. M., & Monahan, R.: Dafny meets the verification benchmarks challenge. In Verified Software: Theories, Tools, Experiments (pp. 112-126). Springer Berlin Heidelberg. (2010)
7. Catano, N., Leino, K. R. M., & Rivera, V.: The eventb2dafny rodin plug-in. In Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on (pp. 49-54). IEEE. (2012)
8. Catano, N., Rueda, C., & Wahls, T.: A Machine-Checked Proof for a Translation of Event-B Machines to JML. arXiv preprint arXiv 1309.2339. (2013)
9. Edmunds, A., & Butler, M.: Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. Programming Language Approaches to Concurrency and Communication-cEntric Software, 1. (2011)